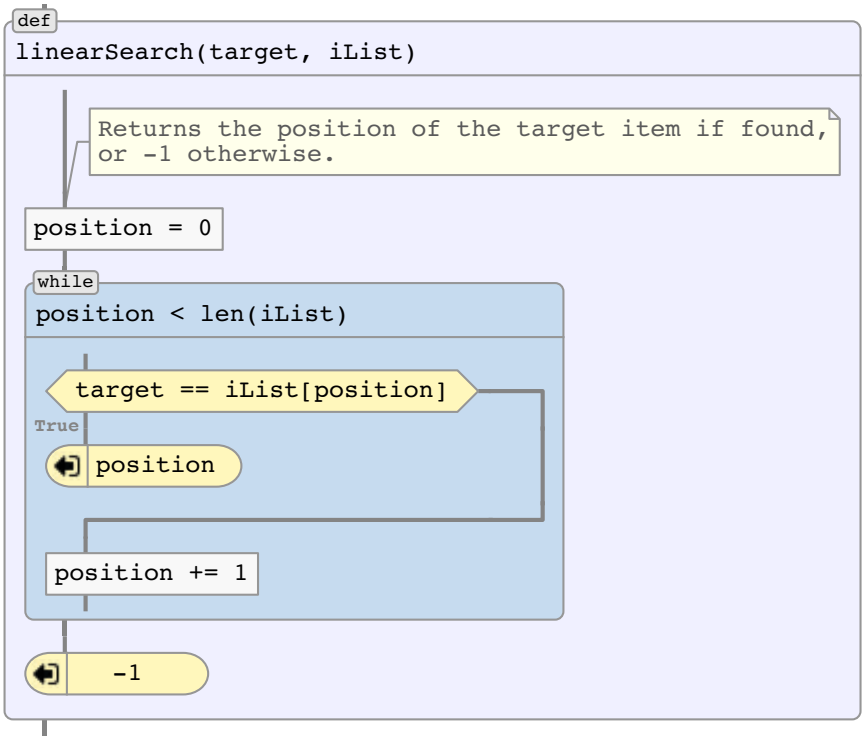
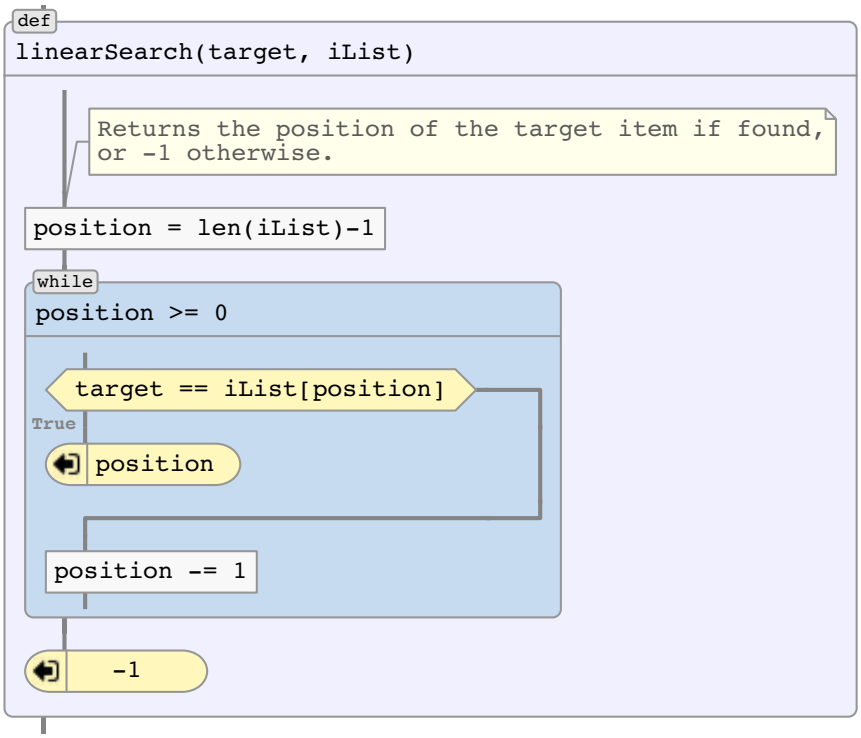
CIND830
Python Programming for Data Science
Lab 7 - Answer Key

1 Search Algorithms

The following algorithm searches a given list sequentially to find the position of a given item. Notably, the performance of that algorithm depends on the location of the target item.



a. Redesign the linear algorithm to start searching from the end of the list rather than from its beginning.



b. Write the code of the algorithm in **1a** then verify its correctness by running the following commands.

```
firstLinearSearch = linearSearch(4, [4, 5, 8, 4, 9, 3, 6, 2])
print(firstLinearSearch)

secondLinearSearch = linearSearch(9, [4, 5, 8, 4, 9, 3, 6, 2])
print(secondLinearSearch)

thirdLinearSearch = linearSearch(2, [4, 5, 8, 4, 9, 3, 6, 2])
print(thirdLinearSearch)
```

```
def linearSearch(target, iList):
    #Returns the position of the target item if found,
    #or -1 otherwise.
    position = len(iList)-1
    while position >= 0:
        # cml 1 sw
```

```
        if target == iList[position]:
            return position
        position -= 1
    return -1

firstLinearSearch = linearSearch(4, [4, 5, 8, 4, 9, 3, 6, 2])
print(firstLinearSearch)        # The result should be 3

secondLinearSearch = linearSearch(9, [4, 5, 8, 4, 9, 3, 6, 2])
print(secondLinearSearch)       # The result should be 4

thirdLinearSearch = linearSearch(2, [4, 5, 8, 4, 9, 3, 6, 2])
print(thirdLinearSearch)        # The result should be 7

fourthLinearSearch = linearSearch(1, [4, 5, 8, 4, 9, 3, 6, 2])
print(fourthLinearSearch)       # The result should be -1
```

c. Compare the results obtained in question **1b** with those of the original linear search that starts the searching process from the beginning of the list.

```
'''
All four test results will be the same,
except for the first one; the output will be 0 rather than 3.
With the modified version of the linear search algorithm,
whenever there are duplicate values, the algorithm will return
the position of the value closer to the end of the list.
However, with the original version, the algorithm will return
the value's position closer to the beginning of the list.
'''
```

d. Which of the tests in question **1b** represents the best case scenario in terms of the `Big O Notation`? Justify your answer.

```
'''
The third test represents one of the best-case scenarios
for the modified linear search algorithm. The computational
complexity of that algorithm, in this case, would be O(1),
which means the algorithm will not change its number
of steps even if the length of the list varies.
'''
```

e. Would the best-case scenario for the modified linear search algorithm be the best scenario for the original one?

```
'''
Notably, the best-case scenario for the modified version of the
linear search algorithm would be the worst for the original one,
unless there are duplicate values. Therefore the third test
represents one of the worst-case scenarios for the original
algorithm with O(n), which is less efficient than O(1).
'''
```

2 Sort Algorithms

The following code defines an algorithm that sorts a list in ascending order.

```
def selectionSort(lyst):
    i = 0
    while i < len(lyst) - 1:          # Do n - 1 searches
        minIndex = i                  # for the smallest
        j = i + 1
        while j < len(lyst):          # Start another search
            if lyst[minIndex] > lyst[j]:
                minIndex = j
            j += 1
        if minIndex != i:             # Swap if needed
            lyst[minIndex], lyst[i] = lyst[i], lyst[minIndex]
        i += 1
    return(lyst)
```

a. Verify the correctness of that algorithm by running the following code and displaying the results.

```
selectionSort = selectionSort([5, 6, 8, 3, 2, 1, 9, 4])
print(selectionSort)
# The output should be [1, 2, 3, 4, 5, 6, 8, 9]
```

b. Which configuration of items results in the smallest number of swaps in the aforementioned algorithm? Which configuration results in the largest number of swaps?

```
''' If all items are sorted in ascending order, there will be no
exchanges or swaps, and the body of the respective one-way statement
will not be executed. However, if the first element is the largest
and the remaining elements are sorted in ascending order, the number
of swaps would be equal to the length of the list minus one.
'''
```

c. Given the following list of items:

```
[71, 6, 35, 24, 61, 82, 27, 1, 17, 83]
```

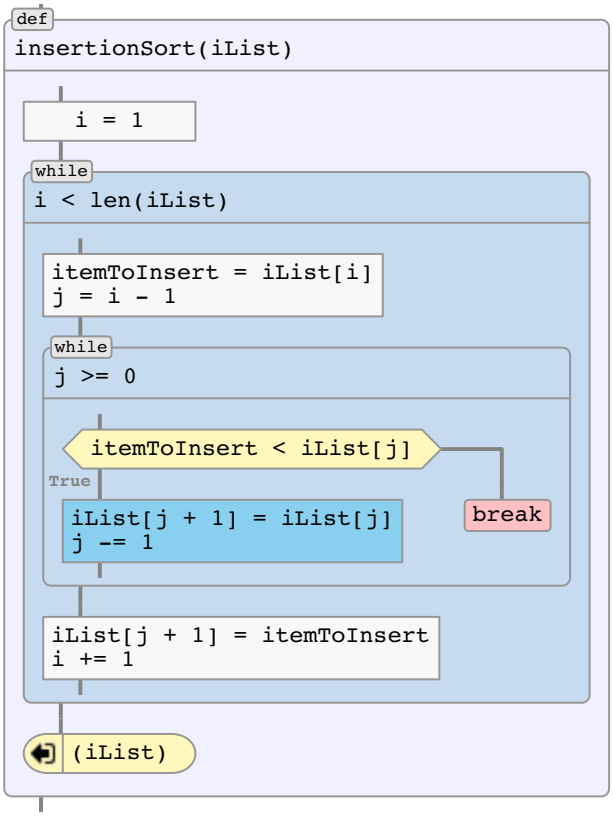
Modify the algorithm to display the contents of the list after each iteration of the outer loop when sorting in ascending order.

```
def selectionSort(iList):
    i = 0
    while i < len(iList) - 1:          # Do n - 1 searches
        minIndex = i                   # for the smallest
        j = i + 1
        print(iList)
        while j < len(iList):          # Start another search
            if iList[minIndex] > iList[j]:
                minIndex = j
            j += 1
        if minIndex != i:              # Swap if needed
            iList[minIndex], iList[i] = iList[i], iList[minIndex]
        i += 1
    return(iList)

selectionSort = selectionSort([71, 6, 35, 24, 61, 82, 27, 1, 17, 83])
print(selectionSort)
#The output should be as follows
'''
[71, 6, 35, 24, 61, 82, 27, 1, 17, 83]
[1, 6, 35, 24, 61, 82, 27, 71, 17, 83]
[1, 6, 35, 24, 61, 82, 27, 71, 17, 83]
[1, 6, 17, 24, 61, 82, 27, 71, 35, 83]
[1, 6, 17, 24, 61, 82, 27, 71, 35, 83]
[1, 6, 17, 24, 27, 82, 61, 71, 35, 83]
[1, 6, 17, 24, 27, 35, 61, 71, 82, 83]
```

```
[1, 6, 17, 24, 27, 35, 61, 71, 82, 83]
[1, 6, 17, 24, 27, 35, 61, 71, 82, 83]
[1, 6, 17, 24, 27, 35, 61, 71, 82, 83]
'''
```

d. The following design represents a different type of sort algorithms, called `InsertionSort`.



Write the code that implements the `InsertionSort` algorithm, then verify its correctness using the following commands:

```
insertionSort = insertionSort([5, 6, 8, 3, 2, 1, 9, 4])
print(insertionSort)
# The output should be [1, 2, 3, 4, 5, 6, 8, 9]
```

```
def insertionSort(iList):
    i = 1
    while i < len(iList):
        itemToInsert = iList[i]
        j = i - 1
        while j >= 0:
            if itemToInsert < iList[j]:
                iList[j + 1] = iList[j]
                j -= 1
            else:
                break
        iList[j + 1] = itemToInsert
        i += 1
    return(iList)

insertionSort = insertionSort([5, 6, 8, 3, 2, 1, 9, 4])
print(insertionSort)
# The output should be [1, 2, 3, 4, 5, 6, 8, 9]
```

e. Given the list in question **2c**, modify the `InsertionSort` algorithm to display the contents of the list after each iteration of the outer loop when sorting in ascending order.

```
def insertionSort(iList):
    i = 1
    while i < len(iList):
        itemToInsert = iList[i]
        j = i - 1
        print(iList)
        while j >= 0:
            if itemToInsert < iList[j]:
                iList[j + 1] = iList[j]
                j -= 1
            else:
                break
        iList[j + 1] = itemToInsert
        i += 1
    return(iList)

insertionSort = insertionSort([71, 6, 35, 24, 61, 82, 27, 1, 17, 83])
print(insertionSort)

#The output should be as follows
```

```
'''
[71, 6, 35, 24, 61, 82, 27, 1, 17, 83]
[6, 71, 35, 24, 61, 82, 27, 1, 17, 83]
[6, 35, 71, 24, 61, 82, 27, 1, 17, 83]
[6, 24, 35, 71, 61, 82, 27, 1, 17, 83]
[6, 24, 35, 61, 71, 82, 27, 1, 17, 83]
[6, 24, 35, 61, 71, 82, 27, 1, 17, 83]
[6, 24, 27, 35, 61, 71, 82, 1, 17, 83]
[1, 6, 24, 27, 35, 61, 71, 82, 17, 83]
[1, 6, 17, 24, 27, 35, 61, 71, 82, 83]
[1, 6, 17, 24, 27, 35, 61, 71, 82, 83]
'''
```

f. Explain why the `InsertionSort` algorithm performs well on partially sorted lists.

```
'''
With partially sorted lists, where at least half of the items are
sorted in ascending order, the insertion locations for items are
found more quickly in the inner loop, and hence, reducing the
amount of executed steps.
'''
```

This is the end of lab 7
Tamer Abdou, PhD